R / Bioconductor: Curso Intensivo

Leonardo Collado Torres
Licenciatura en Ciencias Genómicas, UNAM
www.lcg.unam.mx/~lcollado/index.php

Cuernavaca, México
Oct-Nov, 2008

Introducción y R básico

De donde viene R

- Para muchos R es un dialecto porque es un derivado del lenguaje S creado por John Chambers y co en los Bell Labs. En si, R fue escrito a mitad de los 90s por Ross Ihaka y Robert Gentleman.
- Desde 1997, R ha sido manejado por el *R Development Core Team* y se ha mantenido como open-source.
- Una ventaja de R es que se puede usar en varias plataformas: UNIX, Windows, Mac.
- R en si es un lenguaje de computación creado para facilitar la manipulación de datos, hacer cálculos y gráficas de alto nivel. Es por esto que R es fuerte en estadística.

Propiedades de R

R es un ambiente para trabajar en estadística computacional y al mismo tiempo es un lenguaje de programación. Hay usuarios que solo van a usar las funciones básicas de R (como una calculadora) mientras otros incluso harán paquetes que ligen R con C. En fin, R:

- es efectivo en el manejo de datos y su almacenamiento.
- tiene muchos operadores para hacer cálculos en arreglos (vectores) y matrices.
- tiene una gama de herramientas para el análisis de datos. Hay muchos paquetes disponibles, como la familia de Bioconductor.
- tiene un sistema de gráficas muy útil para el análisis de datos. Excel es cosa del pasado ;)

Propiedades de R

- ya viene con modelos estadísticos.
- hay muchos manuales y un sistema de ayuda bastante bueno. Además hay una comunidad internacional que te puede ayudar :).

Abrir y cerrar R

- Para abrir `R` simplemente tienen que escribir el comando `R` en UNIX. Lo primero que veran es una pequeña descripción de `R` incluyendo la versión que tienen instalada.
- Al abrir `R`, este busca en el directorio donde están información de alguna sesión previa. Esto luego sera útil con los *workspace*.
- Para cerrar `R` simplemente escriban `q()`. Les va a pedir si quieren guardar una imagen del workspace – por ahora digan que no.

Workspace

- Muchas veces tienes que interrumpir tu trabajo. `R` tiene toda una funcionalidad llamada workspace que te ayuda a retomar tu trabajo de sesiones previas.
- Cuando guardas el workspace se crean dos archivos: `.RData` y `.Rhistory` en el directorio donde estes trabajando. Estos almacenan todos los objetos que haya definido el usuario (vectores, matrices, listas, funciones). La próxima vez que abras R en ese directorio, carga todo lo que creaste antes automáticamente.
- Hay una serie de funciones que les pueden ayudar para organizar su trabajo en R. getwd te da tu directorio de trabajo actual, setwd lo cambia, history te muestra los últimos 25 comandos que usaste y `history(max.show=Inf)` te muestra todos.

Workspace

- Otras funciones útiles son savehistory y loadhistory. Además si quieres ver que objetos tienes en tu sesión puedes usar objects o ls; puedes quitar objectos con rm.
- Si quieren guardar el workspace manualmente o con un nombre diferente a `.Rdata` usen `save.image()`. De igual forma, pueden cargar un workspace manualmente con load.
- Para checar las opciones del ambiente de `R` usen options. Por ejemplo, pueden cambiar cuantos dígitos se imprimen en el output.

R como Calculadora

- R es un *expression language*. Aka[1], una R no es igual a una r. Los nombres de variables tienen que empezar por un punto[2] o caracteres alfanuméricos.

```
> 2 + 2
> 2^2
> r <- c(1:3, 4.5, 109)
> pi * r^2
> sqrt(36)
> sin(2 * pi)
> exp(1)
> log(10)
> log(10, base = 10)
```

[1] also known as

[2] Una letra le tiene que seguir al punto para que sea un nombre válido

Asignación de valores

- En R, hay 3 formas de asignar valores, aunque en general se usan solo dos: = y <-
- Preferencialmente usen <- simplemente para evitar confusiones. Es que el signo = se usa para el paso de valores en las funciones.

```
> A <- c(a = 1, b = 2)["b"]
> A = c(a = 1, b = 2)["b"]
> A

b
2
```

- Aquí queda más clara la asignación en la primera línea, aunque las dos hacen lo mismo.

Vectores

- `R` es un lenguaje vectorizado, así que puedes ver todas tus variables como vectores. Hay varios *modos*: `numeric`, `character`, `logical`.
- Tal vez la función más usada en `R` es `c()`. Con esta función puedes generar vectores de datos.

```
> v1 <- c(1:10)
> v2 <- runif(10)
> v3 <- sample(c("A", "C", "G", "T"),
+     size = 10, replace = TRUE)
> v4 <- v3 %in% c("A", "G")
> v5 <- c("foo", 2, TRUE)
> v6 <- c(2, "3")
```

Vectores

- Puedes usar la función mode para encontrar que tipo de vector tienes. Además, con as puedes cambiar el modo de un vector. Intenten cambiar al modo *numeric* los vectores `v5 y v6`:

```
> as.numeric(v5)
> as.numeric(v6)
```

Un ejemplo sencillo

Example (Tam. Fagos)

Conocemos el tamaño del genoma de 10 bacteriofagos. Queremos explorar esta información. Sus tamaños en mbs son: `233.2 180.5 280.3 244.8 252.4 178.2 211.2 196.2 176.8 185.7` Almacenen esta info en el vector `fagos` y encuentren:

1. La suma de los tam. de los genomas
2. La longitud del vector fagos
3. El tam. promedio de los genomas

Así se resuelve:

Un ejemplo sencillo


```
> fagos <- c(233.2, 180.5, 280.3,
+     244.8, 252.4, 178.2, 211.2,
+     196.2, 176.8, 185.7)
> sum(fagos)
[1] 2139.3
> length(fagos)
[1] 10
> sum(fagos)/length(fagos)
[1] 213.93
> mean(fagos)
[1] 213.93
```

Además, pueden usar `sort()`, `min()`, `max()`, `range()`, `diff()`, `cumsum()` y summary().

Reciclaje de vectores

- En `R` la mayoría de las funciones están vectorizadas[3]. Por ejemplo cuando hacemos `x = 2; y = 3; x + y` en realidad estamos haciendo $x[i] + y[i], i \in 1, \ldots \max\{|x|, |y|\}$
- Si la longitud de los dos vectores no es la misma, `R` recicla el más chico con tal de llegar a la longitud del grande.
- Prueben con `c(2,3) + c(3,4,5)` y compárenlo con `c(2,3) + c(3,4,5,8)`
- Siempre tengan cuidado con los warnings que salen. En el caso del reciclaje, estos solo salen si `(length(x) %% length(y)) != 0`
- Con esto en mente ahora podemos encontrar la varianza de un vector de forma *manual*:

Reciclaje de vectores

Example (Varianza)

```
> x <- c(2, 3, 5, 7, 11)
> xbar <- mean(x)
> n <- length(x)
> sum((x - xbar)^2)/(n - 1)
[1] 12.8
> var(x)
[1] 12.8
```

Supongo que ahora solo usarán `var()` ;)

[3] En las que no, es porque no tendría sentido vectorizarlas

Buscando ayuda

- Tal vez lo más importante en cualquier lenguaje de programación es saber donde buscar ayuda. `R` tiene un sistema de ayuda bastante completo, aunque a veces si hay que meterse a google.
- La función madre para buscar ayuda es `help()` Digamos que no saben que hace la función `names`, así que pueden buscar ayuda al respecto con `help("names")` o alguno de los atajos: `?names` o `?"names"`.
- Si no saben que es lo que buscan pueden usar `help.start()` que abre una página html. Aquí siempre encontrarán ejemplos que los pueden ayudar a entender. Estos los pueden copiar y pegar en R para correlos :)

Buscando ayuda

- Para hacer una búsqueda más profunda usen `help.search()` ya que esta función busca en más secciones de los manuales de ayuda. Por ejemplo, `help.search("names")`
- Si están buscando nombres de funciones, usen `apropos()` Por ejemplo, `apropos("names")`. Otras funciones útiles son `RSiteSearch()`, `args()` y `example()`.

Usando las herramientas de ayuda

Example (data.entry y names)

En este ejemplo apliquen lo que acabamos de ver para buscar ayuda sobre `names` y `data.entry`. Luego:

1. Creen el vector `simpsons` con los nombres de los 5 protagonistas de los Simpons usando la función `c()`.
2. Usando `names` cambien el vector `simpsons` para que refleje quien es el padre, madre, etc.
3. Usen `data.entry` y agreguen a dos amigos de la familia al vector `simpsons`.
4. Completen-actualizen los nombres del vector `simpsons`

Un ejercicio simple de tarea

Aprendiendo a hacer secuencias y repeticiones

Como un ejercicio simple de tarea quiero que aprendan a usar las funciones `seq`, `rev`, `paste` y el operador `colon :`. Almacenen en diferentes vectores los siguientes datos sin usar `c()` a menos de que no haya otra opcion.

- Los números de Fibonnacci del 1 al 34
- Las fracciones 1/1 hasta 1/10 usando enteros en el denominador.
- Los años desde 1964 hasta el 2008.
- Los múltiplos de 25 desde 1000 hasta 0 *en ese orden*.
- La serie "A" "A" "T" "T" "T" "T" "C" "G" y luego conviertanla a "AATTTTCG".

Usar los índices

- En R al igual que en otros lenguajes es importante aprender como accesar a la información que tienes en tu variable; vectores en este caso.
- Muchas veces van tener su información almacenada en un vector de datos; hay cuatro índices principales que puedes usar para seleccionar subconjuntos de tu vector: vectores lógicos, un vector de enteros positivos, otro de negativos y un string de caracteres.

Vectores Lógicos

- Cuando accesas a un vector por medio de un vector lógico, estas filtrando a los que te dan como `TRUE`[4] en alguna comparación.

```
> x <- c(-2:2)
> y <- x/x
> z <- y[!is.na(y)]
> z2 <- y[!is.na(y) & x > 0]
```

- En la primera `z`, estamos eliminando a los valores `NaN`[5] como sería 0/0. En `z2` además queremos solo los de `x>0`.
- Tengan en mente que la longitud de los vectores `z` y `z2` son diferentes a la longitud de `x`.

Vectores Lógicos

Operadores Lógicos

En R hay diversos operadores lógicos que funcionan como en otros lenguajes con alguna pequeña diferencia. Corran los siguientes comandos para aprender como funcionan :). Para un aprendizaje más detallado lean la ayuda de `?">"` y `?all.equal`

```
> x <- c(1:5)
> x < 5
> x > 1
> x > 1 & x < 5
> x > 1 && x < 5
> x > 1 | x < 5
> x > 1 || x < 5
> x == 3
```

Vectores Lógicos


```
> x != 3
> !x == 3
> x == c(2, 4)
> x %in% c(2, 4)
```

[4] Muchas veces puedes usar `T` en vez de `TRUE`; eviten llamar una variable como T

[5] NaN significa Not a Number

Vector de enteros positivos

- Tal vez la forma más comun de accesar a un vector de datos es por posición. Aquí simplemente las posiciones van desde 1[6] hasta n donde n es la longitud del vector de datos.
- Si `x` tiene 100 elementos, puedes entrar a los primeros 10 usando `x[1:10]` o a los elementos 1, 5 y 8 usando `x[c(1,5,8)]`.
- Otra forma de usar esta tipo de índice sería:

```
> c("A", "T", "C", "G")[rep(c(1,
+     2, 2, 4, 3), times = 2)]
```

[6] Es diferente de Perl!

Vector de enteros negativos

- En realidad esto es muy sencillo. Simplemente son las posiciones que queremos excluir.
- En el siguiente ejemplo simplemente nos quedamos con las posiciones 1, 7, 8 y 10.

```
> x <- c("inicio", rep(c("A", "T",
+     "C", "G"), times = 2), "fin")
> y <- x[-c(2:6, 9)]
> y
[1] "inicio" "T"      "C"      "fin"
```

Por vector de caracteres

- En realidad esta forma es muy parecida a las anteriores. Simplemente tienen que poner entre comillas dobles las palabras que identifican a las posiciones.

```
> fagos <- c(233.2, 180.5, 280.3)
> names(fagos) <- c("Aeromonas phage Aeh1",
+     "Enterobacteria phage RB43",
+     "Pseudomonas phage phiKZ")
> fagos["Aeromonas phage Aeh1"]
> fagos[grep("Aeh1", names(fagos))]
```

Chequen la función grep! `which` también es bastante útil.

- Para ahora ya se deben haber dado cuenta... las funciones siempre usan (...) y los vectores de datos usan [...]

El famoso "if"

R ofrece las estructuras de control más clásicas con lo cual luego podremos hacer funciones.

- El `if` es la estructura más simple y su sintaxis es bastante sencilla: `if (cond1=vdd) {cmd1} else {cmd2}`
- El `ifelse` no se diferencia tanto, aunque es una función. Mas bien es como en Excel; su sintaxis es: `ifelse(prueba, valor-vdd, valor-falso)`
- Aquí les muestro un par de ejemplos:

```
> if (1 == 0) {
+     print(1)
+ } else {
+     print(2)
+ }
```

El famoso "if"


```
[1] 2
```

```
> x <- 1:10
> ifelse(x < 5 | x > 8, x, 0)
```

```
 [1]  1  2  3  4  0  0  0  0  9 10
```

Ciclos

- El `for` ya no es tan similar a lo que conocemos. Su sintaxis base es: `for(variable in secuencia) {comandos}`
- El segundo tipo de ciclo más usado es `while`. Su sintaxis base es: `while(condición) {comandos}`
- El último y que casi nunca se usa es el `repeat`. Realmente no se los recomiendo... en fin, aquí tienen un ejemplo de un for:

```
> x <- 1:10
> z <- NULL
> for (i in 1:length(x)) {
+     if (x[i] < 5) {
+         z <- c(z, x[i] - 1)
+     }
```

Ciclos


```
+     else {
+         z <- c(z, x[i]/x[i])
+     }
+ }
> z
 [1] 0 1 2 3 1 1 1 1 1 1
```

Matrices

- R te permite tener variables de tipo matriz. Estas simplemente son vectores con un vector dimensional que es diferente de NULL.
- Si le cambias el vector de dimensiones a un vector, lo puedes volver una matriz[7]. Esto afecta como se imprime como ven a continuación:

```
> V <- runif(100)
> print(V[1:9])
```

```
[1] 0.4479711 0.7877175 0.7089860
[4] 0.8446513 0.3021317 0.5532824
[7] 0.6777822 0.2457244 0.8265174
```

```
> dim(V)
```

Matrices


```
NULL
```

```
> dim(V) <- c(2, 5, 10)
> print(V[, , 1])
```

```
          [,1]      [,2]      [,3]
[1,] 0.4479711 0.7089860 0.3021317
[2,] 0.7877175 0.8446513 0.5532824
          [,4]      [,5]
[1,] 0.6777822 0.8265174
[2,] 0.2457244 0.9948602
```

[7] No a fuerzas es de 2 dimensiones

Álgebra Matricial

- R te permite hacer operaciones entre matrices como:
  - La multiplicación con `%*%`
  - El inverso X^{-1} con `solve`
  - La transpuesta X^t con `t`
  - El producto *exterior* de dos vectores (xx^t) con `%o%`
  - El productor Kronecker de dos matrices con `%x%`
  - A^tX y AX^t con `crossprod` y `tcrossprod`
  - La descomposición eigen de una matrix con `eigen`
- Si quieren aprender más sobre álgebra matricial vayan a wikipedia.

Multiplicación de Matrices

- Para aprender un poco de la multiplicación de matrices corran los siguientes comandos y comparen los resultados:

```
> X <- 1:16
> dim(X) <- c(4, 4)
> X
> X * X
> X %*% X
> sum(X[, 1] * X[1, ])
> sum(X[, 1] * X[2, ])
```

Multiplicación de Matrices

Índices en matrices

De paso ya aprendieron a usar índices en matrices :)! Ahora ya saben como recuperar alguna columna o línea de una matriz; por ejemplo, la línea `1` con `X[1,]`. Recuerden que el primer índice son las líneas (m) y el segundo son las columnas (n) en el caso de una matriz `m x n`.

Leer un archivo

- Algo esencial que todos sepan es como abrir un archivo o directorio en `R`. Dudo mucho que quieran usar `scan()` y llenar los datos manualmente :P.
- `R` puede manejar varios archivos con números para una sola variable, tablas de números, archivos tipo `csv` y más. Por ejemplo, podríamos haber leido la info de los fagos así: `fagos <- scan(file="fagos.txt")`[8]
- Las funciones principales para leer archivos son `scan()`[9], `read.table()`, `read.csv()` y `source()`.
- Si quieren especificar el archivo de entrada cuando ejecuten el comando, pueden usar `read.table(file=file.choose())`.

[8] El archivo tendría que estar en el mismo folder donde estamos trabajando

[9] Especificando el archivo de entrada

Data Frame

Data Frames

Un formato muy usado en `R` son los *data frames*. Estos en realidad son como una hoja de cálculo donde cada columna es una variable. Pueden accesar a cada columna con `dataframe$variable` o `dataframe[["variable"]]`. Además pueden usar las funciones `attach` y `detach` para agregar las variables de un `data frame` al ambiente de `R`.[10]; la función `with(data.frame, comando)` hace lo mismo. Finalmente, pueden ver el principio o el final de un `data frame` usando `head()` o `tail()`.

[10] No es recomendable si piensan modificar los valores del `data frame` o si ya tienen variables con los mismos nombres

read.table

Example (Leer una tabla)

Para leer una tabla con algo de info sobre unos fagos usen:

```
> arch <- "../../data/10biggestPhages.txt"
> fagos.gr <- read.table(file = arch,
+     header = TRUE)
```

R también te permite leer archivos que están en servidores web. Esta misma tabla también la pueden leer así[11]:

read.table


```
> sitio <- "http://kabah.lcg.unam.mx/~lcollado/R/"
> sitio <- paste(c(sitio, "data/10biggestPhages.txt")
+     collapse = "")
> fagos.gr <- read.table(file = url(sitio),
+     header = TRUE)
> fagos.gr <- read.table(file = sitio,
+     header = TRUE)
> fagos.gr[c(2:4)]
```

	GenomeSize	EMBL	Taxid
1	280334	AF399011_GR	169683
2	252401	AY939844_GR	268746
3	244834	AY283928_GR	75320
4	233234	AY266303_GR	227470
5	211215	AJ697969_GR	273133

read.table


```
6       196280 AJ630128_GR 238854
7       185683 AP008983_GR  12336
8       180500 AY967407_GR 115991
9       178249 AY940168_GR 268747
10      176847 DQ149023_GR 382359
```

```
> fagos.gr$Taxid[2:3]
```

```
[1] 268746  75320
```

- Noten que las 2 formas de `read.table` son iguales, solo que una es más clara en su sintaxis. Además, el output de `read.table` es un `data frame`.
- Chequen los argumentos de la función `read.table`; en especial `sep` y `header`.

[11] Mejor usen `sitio <- "http://kabah.lcg.unam.mx/ lcollado/R/data/10biggestPhages.txt"` – tuve que hacerlo de otra forma por el espacio

Directorios

- Muchas veces quieres abrir más de un archivo de un directorio o folder. Tal vez no quieres abrir todos, así que tienes que buscar un patrón en sus nombres.
- La forma más automática de hacerlo es así:

```
> setwd("../../data")
> files <- list.files(pattern = "s.txt$")
> for (i in files) {
+     x <- read.table(i, header = TRUE)
+     assign(i, x)
+     print(i)
+ }

[1] "10biggestPhages.txt"
[1] "fagos.txt"
```

Funciones: lo básico

- Algo muy importante es que puedan definir sus propias funciones. En si las funciones son objetos con el modo *function* y muchas funciones base como `mean` están definidas de la misma forma que las que ustedes harán.
- La sintaxis basíca para definir una función y llamarla son:

```
> mifun <- function(arg1, arg2, ...) {
+     lo_que_sea
+ }
> mifun(arg1 = ..., arg2 = ...)
```

- Les recomiendo ampliamente que especifiquen valores *default* a sus argumentos. Esto lo hacen con `arg1 = val.def`.

Funciones: lo básico

- Los nombres de las funciones siguen las mismas reglas que las de los objetos(variables). Tengan cuidado de no repetir nombres de funciones ya existentes en su ambiente de R.
- Cuando llaman a una función, pueden poner los argumentos en el mismo orden que los definieron o usar sus *tag* (nombres).
- Tomen nota de que un valor modificado adentro de una función es modificado temporalmente. Si en una función quieren modificar un valor externo, tienen que usar el operador `<<-` o la función `assign`.
- En cuanto al cuerpo de la función, pueden tener un solo comando o muchos.[12] El valor que regresa una función es el último en ser evaluado, o el que especifiquen con `return`.

Funciones: lo básico

Aquí les ejemplifico un poco lo que acabamos de ver:

```
> fact <- function(x = 1) {
+     ret <- 1
+     for (i in 1:x) {
+         ret = ret * i
+     }
+     return(c(x, ret))
+ }
> fact()
[1] 1 1
> fact(x = 5)
[1]   5 120
> fact(6)
```

Funciones: lo básico


```
[1]   6 720
```

[12] Si tienen uno no necesitan usar las llaves.

Funciones: ... y missing

- Cuando no espeficicas los argumentos, lo que hace `R` es buscar a cual corresponden.[13]
- Algo importante es poder pasar los argumentos de una función a otra (los que compartan). Para eso se utiliza el argumento especial `...`
- Finalmente, usen la función `missing` para checar si un argumento no fue proporcionado por el usuario.

Un ejemplo con `missing`:

Funciones: ... y missing


```
> mifun2 <- function(x1 = 5, arg.op) {
+     if (missing(arg.op)) {
+         z1 <- 1:5
+     }
+     else {
+         z1 <- arg.op
+     }
+     return(z1/x1)
+ }
> mifun2(x1 = 5)

[1] 0.2 0.4 0.6 0.8 1.0

> mifun2(x1 = 5, arg.op = 30:25)

[1] 6.0 5.8 5.6 5.4 5.2 5.0
```

[13] Para entender mejor vean `match.arg`

Funciones: control

- Algunas de las funciones que harán en el futuro requerirán de las utilidades de control. En `R` hay 3 de estas: `return`, `stop` y `warning`.
- El `return` ya lo usamos. Este especifica que valores regresa la función además de terminarla.
- El `stop` para la función e imprime un mensaje de error.
- Finalmente, el `warning` simplemente imprime un mensaje pero no para la función. Aquí les muestro un ejemplo para que lo corran:

Funciones: control


```
> mifun3 <- function(x1) {
+     if (x1 > 0) {
+         print(x1)
+     }
+     else if (x1 == 0) {
+         warning("El valor debe ser > 0")
+     }
+     else {
+         stop("Hay un error porque x1 < 0")
+     }
+ }
> mifun3(x1 = 0)
> mifun3(x1 = -2)
```

Funciones: nuevos operadores

- En R puedes definir un tipo de función especial, tal como %in%. Estos operadores tienen que ser binarios. Se definen y usan con la sintaxis:

```
> "%nombre%" <- function(x, y) {
+     lo_que_sea
+ }
> datos1 %nombre% datos2
```

- Noten el uso de las comillas dobles en la definición del operador.

Paquetes

- No sé si han intentado instalar un paquete de `BioPerl` o algo parecido... bueno, en `R` es mucho más sencillo y no hay tantos problemas entre las diferentes plataformas.
- En si, no pueden instalar paquetes en los servidores ya que no son admins, pero si es importante que aprendar como hacerlo.
- Hay dos fuentes principales de paquetes para `R`. Una es `CRAN` y la otra `Bioconductor`[14].
  - CRAN[15]
  - Bioconductor
- Cada paquete tiene su ayuda, y pueden ver la descripcion basíca, sus dependencias y la lista de funciones del paquete con:

Paquetes


```
> help(package = nombre.paquete)
```

- En general, hay una página para cada paquete donde te explican como usarlo y te muestran ejemplos.
- Les recomiendo que escriban lo siguiente para explorar lo que te ofrece R de base:

```
> help(package = base)
```

[14] veremos más al respecto en otra clase

[15] De aquí incluso pueden bajar R para su laptop

Paquetes: instalación

- Para ver que paquetes tienen instalados usen la función `installed.packages()`
- Para ver que paquetes hay disponibles en un *mirror* usen `available.packages()` y para instalar alguno usen `install.package()`
- Finalmente, para los de `Bioconductor` tienen que usar un script de `R` llamado `biocLite`. Esto lo hacen con `source`.

Aquí tienen un par de ejemplos:

```
> install.package("mcmc")
> source("http://bioconductor.org/biocLite.R")
> biocLite("GeneR")
```

Paquetes: para usarlos

- Los paquetes generalmente traen funciones. Pero también pueden ser librerías de datos. En el caso de `Bioconductor` hay toda una sección para bajar datos experimentales.
- Acuérdense que el ambiente de `R` carga las funciones base. Para poder usar las funciones o la info de un paquete tienen que usar la función `library()`.
- Si luego se les olvida que paquetes tienen cargados en su sesión, usen `search()`.

Paquetes: para usarlos

Example (GeneR)

Aquí les muestro un ejemplo donde usamos el paquete `GeneR`.

```
> library(GeneR)
> s <- "atuuutututu"
> placeString(s)
> dnaToRna()
> getSeq()
> rnaToDna()
> getSeq()
```

El resultado es `[1] "ATTTTTTTTTT"`

Fin de la clase

- Por último quiero que hagan el ejercicio 1 que pueden encontrar en la página del curso.
- Suerte! :)